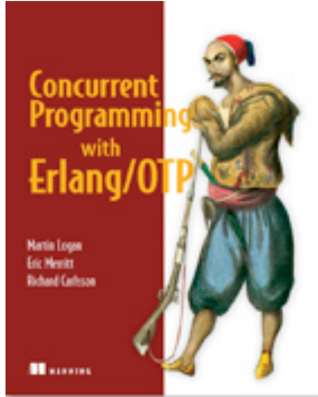
MANNING PUBLICATIONS

# OTP Applications

Excerpted from




Erlang and OTP in Action
EARLY ACCESS EDITION
Martin Logan, Eric Merritt, and Richard Carlsson
MEAP Release: August 2008
Softbound print: Summer 2010 (est.) | 500 pages
ISBN: 9781933988788
Manning Publications

*This article is taken from the book* ***Erlang and OTP in Action.***

In this article we will talk about how to package up your code and get it to fit nicely into a normal OTP system. For some reason this topic tends to cause a lot of confusion to folks who are new to the system. I can understand that. When I first started working with Erlang, what is traditionally known as OTP was black magic, poorly documented and with very few examples. After a lot of trial and Error, and quite a few helpful hints from various old timers I was able to figure it out. Fortunately for you and I both, once you get your mind around the basic concepts it's really pretty simple. Just remember that Applications are living things with explicit startup and shutdown semantics. Applications must have a root supervisor whose job is to manage the processes that make up the Application.

## Applications

Creating an Application consists mostly of setting up your directory structure and including a bit of simple metadata. This metadata tells the system details; it needs to know how to start and stop the application. It also details information about what order to start the Applications in. There is some coding involved as well but we can get to that in a minute.

**The Drawbacks of Naming Them ''**

The main drawback to packages en Erlang is that they are not called packages, jars, gems or anything else that would be explicit and understandable. They are called 'Applications' instead. In one way it makes sense, but it also tends to cause a huge amount of confusion. When people are talking about Applications you never know whether they are talking about them in the Erlang sense or the broader sense used by the rest of the industry. To reduce the level of ambiguity, from here on out when we are talking about Applications we will always mean Applications in the Erlang/OTP sense.

Erlang, in general, doesn't have an archived format for its Applications. It just uses a very simple directory layout. The directory structure for the Application is shown below.

```
<application-name>-<application-version>
  |
  |- ebin
  |- include
  |- priv
  |- src
```

That's it! That is most of what you need to do to get a valid format for your Application. You should, of course, replace <application-name> with the name of your Application. In our case, it's telnet_server. You should also replace <application-version> with the version of your application. For telnet_server I am going to choose 0.1.0. Each directory listed here is a place for something. Some of them are self-explanatory others need a bit more detail. Take a look at table A.

Table A Application Directories and Their Importance

| Directory | Description |
| --- | --- |
| **ebin** | This is the location where all of your complied code should end up.  Its also the location of the dotapp file, which we will talk about in just a bit. |
| **include** | This is where your public header files go. Basically any hrl file that is part of your public API should end up in this directory. Private hrl files that are only used within your code and are not intended for public consumption should end up in the same place as all the rest of your source. |
| **priv** | Ah, the priv dir. Unless you have already been using Erlang for a bit you probably have no clue what goes here. Never fear its actually pretty simple. Anything that needs to be distributed with your application ends up in this directory. This ranges from templates to shared objects and dlls. The location of an application's priv directory is very easy to access. Just call the function code:priv_dir(<application-name>). This will return the full path of the priv dir as a string. |
| **src** | I really hope you where able to guess this one. If not it this may not be the right book for you. This is where you put all the source related to your application. All of your Erlang, ASN.1, YECC, MIB etc. It all goes here.  Its not required that you distribute your source code with your application. However, it's a very common thing to do is and is generally expected. |

Now that we have our source in the layout that Erlang expects, we can work on adding the metadata that the system expects. It's just a few Erlang terms that describe the application in a file called <application-name>.app in the ebin directory. In Listing A we'll make one up for our telnet_server.

## Listing A Application Metadata

```
%% -*- mode: Erlang; fill-column: 75; comment-column: 50; -*-

{application, telnet_server,
 [{description, "Telnet server for Concurrent Programming with Erlang/OTP"},
  {vsn, "0.1.0"},
  {modules, [telnet_server,
             ts_sup,
             ts_app]},
  {registered, [ts_sup]},
  {applications, [kernel, stdlib]},
  {mod, {ts_app, []}}]}.
```

This file is used by the OTP release handler to understand how your Application fits into an overall release. To understand how this works you should understand that Applications in Erlang are quite a bit different then what you are used to in other languages. In Java jars or ML modules, the code is dead, meaning that it doesn't do anything until it is called by some running code. In Applications this isn't the case at all. The runtime system starts the Application when the system starts and stops it when the system is shut down. This makes Applications a living system and requires that the system understand where your application fits and when and how to start it.

The format of this file itself is pretty simple. It's just an Erlang term terminated by a period. It consists of three elements. The first element is the atom 'application'. The second term is the name of the application. In this case that's 'telnet_server'. The final element in the tuple is an association list of parameters, some of which are required others which are not. The ones I have included here are the basic ones that you need in most applications. Depending on the nature of your application you may or may not need some of these parameters. For now I will give you the information you need to get off the ground. Lets take a second and go over these parameters, listed in table B.

Table B*.app File Parts

| Part | Description |
|---|---|
| **description** | This is exactly what you think. It's a short description of your application. Usually just a sentence or two, though it can be as long as you would like. |
| **vsn** | This describes the version of your application. The version can be anything you would like. Its formatted as a simple string. However, I suggest that you try to keep your versions to the normal <major>.<minor>.<patch>. Erlang and OTP wont have any problem with any type of version you put there. There are systems out there that try to parse this and they may not be as well written. There is also the advantage that everyone out there understands this format and may not understand your personal version format. In the end its up to you. |
| **modules** | This is a list of all the modules in your application. It's a bit tedious to maintain, but there are tools out there to help you with that maintenance. This is a simple list order doesn't matter at all. Just stick the names in there as you need them. |
| **registered** | This is a bit more interesting then the others. As you know Erlang has the concept of registered processes. These are long lived processes that are registered at under an addressable name. They function like little services in your Application. OTP needs to |

| | |
|---|---|
| | know which process are registered for various purposes such as system upgrades. So this just lists the registered names of processes in the system. |
| **applications** | This is a another interesting one. Applications have dependencies. Applications, because the are living systems, expect these dependencies to be started and available when it starts. This entry is a list of all the applications that need to be started before this application can be started. Order doesn't matter here either. OTP is smart enough to look at the entire system and understand what needs to be started when. |
| **Mod** | This is another artifact of the concurrent nature of an Erlang system. When an application is started by the runtime system there must be code that is run. What this actually does depends quite a lot on what your Application needs to do. OTP has a specialized behaviour for Application startup. That behaviour is the 'application' behaviour (ok, naming hasn't been one of Erlangs strengths). There should be one module that implements this behaviour in every Application. This parameter contains the name of that module. |

There are other parameters that are available here for your use. We wont go into them now because they aren't really relevant to our telnet_server app. We have the directory structure set up. We have the metadata in place. However, we haven't quite done everything we need to do to package up our telnet server. The last bit that goes into making an Application is the Application Behavior.

## *The Application Behavior*

Every Application needs one module that implements the 'application' behavior. This behavior provides the start up logic for the system. Every application starts up the root supervisor, it may or may not do other things depending on the needs of your system. Listing B jumps into the behavior.

**Listing B Application behaviour**

```
-module(ts_app).

-behaviour(application).

-export([
     start/2,
     stop/1
     ]).

start(Type, StartArgs) ->
    case ts_sup:start_link(StartArgs) of
    {ok, Pid} ->
        {ok, Pid};
    Error ->
        Error
    end.

stop(State) ->
    ok.
```

The 'application' behavior for our telnet server is pretty normal for an application. I have taken out most of the documentation so we can get a nice overview of the code itself. We start out with the normal header. We then specify that this implements the application behavior via '-behavior' pragma. It is not absolutely necessary that you provide this pragma, it mainly exists so that the Erlang compiler can check that you actually implement the functions required by the behavior. During runtime whether or not you implement the behavior is ignored. The functions are just called and if they don't exist an error is thrown. So implement the behavior even though you don't actually need to. It will save you quite a bit of trouble in the future for no real cost.

Next we export the functions that we have implemented. In general we use to export pragma's, one for the API of our module and one for the functions required by the behavior that we are implementing. Here we are only really implementing a behavior so we only have the one export.

The first real function that we care about is the start function. This is where the magic happens. The start function will be called when your Applications starts up. It expects the Pid of the root supervisor to be returned. So at the very least you need to start up the root Supervisor. You can, of course, do anything else you would like to do here as well. For telnet_server all we really need to do is start the root supervisor. You can see that we do that with the ts_sup:start_link function. We check the return value to make sure it works for us and then we return the value. If we don't get the value we expect we return an error.

Well we have implemented our Behavior but now we hit on a small issue. What do we name our module? This is an important topic because all modules in Erlang live in a single namespace. This has certain implications for module naming and means that this module naming takes a bit more then the usual amount of thought.

#### Naming

One other thing you may have noticed in our examples is that we have been using modules named ts_*. There is a reason for this. Erlang has a flat namespace. This means that every module exits in the same global namespace. While this makes things simple it does open up the possibility of namespace collisions. These are always nasty little problems to debug and there really isn't a lot you can do about them when they occur, unless one of the modules colliding is your own. Unfortunately, in practice, this doesn't usually happen. It’s almost always two modules in different third party libraries. In the interest of reducing the probability of these types of namespace collisions the community has adopted the practice of building a prefix into the module names. This is very similar to what has been done in other languages that lack a namespace. It’s a bit inelegant but in general it works quite well.

> **Prefix Naming**
>
> The authors typically choose the acronym one would form from the name of an application as the prefix for each of its modules. Telnet Server would be T.S hence ts_.

In practice the namespace consists of the initials of the application. It is separated from the actual module name by an underscore. We do this for all of our modules. For a few we have some standard names we use. For example, for the module that implements the 'application' behavior we usually name it <namespace>_app and for the root supervisor we usually name it <namespace>_sup. For our application it means that we are going to have at least two modules, one called ts_app and another called ts_sup. Ts_app is the module that implements the 'application' behavior while ts_sup is our root supervisor.

The modules ts_app and ts_sup aren't really very useful without something that actual does something. We will take that module and rename it from telnet_server to ts_server.

So there are three things that we need to do to package up the Application.

1. Use the correct directory structure.

2. Add the Application metadata in the form of the *.app file.
3. Create a module that starts our Application and implements the Application Behavior.

Of course there are a few other things that need to happen. We also need to create a root supervisor. We also need to make sure that all of our supervised processes implemented are supervisable. At our current level of knowledge this basically means that they implement one of the OTP behaviors like gen_server, gen_event, gen_fsm, etc.

Well, that just about wraps up the salient points of OTP Applications.

For Source Code, Free Chapters, the Author Forum and more information about this title go to
http://www.manning.com/logan/